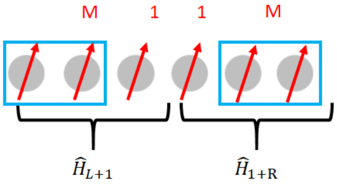
Part 1: Basic algorithm

The basic algorithm goes like this:

1. consider the biggest system size N that can be diagonalized exactly with reasonable resources and regroup the N sites in two single sites in the middle and others in two groups of M sites as shown below:



The system's Hamiltonian can be written as

$$\widehat{H}_N = \widehat{H}_{L+1} + \widehat{H}_{\text{int}} + \widehat{H}_{1+R}$$

where $\widehat{H}_{L+1}$ and $\widehat{H}_{R+1}$ in the literature are referred at as the left and right enlarged blocks, and $\widehat{H}_{\text{int}}$ is the interaction among them.
The dimension of the Hilbert space of the system is (dm)^2, where d is the single site physical dimension and m=d^M the dimension of the M grouped sites.

The diagonalization of $\widehat{H}_N$ returns the ground state expressed in the basis

$$\left|E_0^N\right\rangle = \sum_{\beta_L\beta_R} \psi_{\beta_L\beta_R}\left|\beta_L\beta_R\right\rangle$$

where $|\beta_i\rangle$ spans the basis of the left and right half sites β_i=1,…,d^{M+1}

2. compute the density matrix of the ground state

$$\rho_0^N = \left|E_0^N\right\rangle\left\langle E_0^N\right| = \sum_{\beta_L\beta_R}\sum_{\beta_L'\beta_R'} \psi_{\beta_L\beta_R}\psi^*_{\beta_R'\beta_L'}\left|\beta_L\beta_R\right\rangle\left\langle\beta_R'\beta_L'\right|$$

and the reduced density matrix of one half of the system

$$\rho_L = tr_R(\rho_0^N) = \sum_{\beta_R''}\left\langle\beta_R''\right|\Big(\sum_{\beta_L\beta_R}\sum_{\beta_L'\beta_R'} \psi_{\beta_L\beta_R}\psi^*_{\beta_R'\beta_L'}\left|\beta_L\beta_R\right\rangle\left\langle\beta_R'\beta_L'\right|\Big)\left|\beta_R''\right\rangle$$

$$= \sum_{\beta_L\beta_L'}\Big(\sum_{\beta_R}\psi_{\beta_L\beta_R}\psi^*_{\beta_R\beta_L'}\Big)\left|\beta_L\right\rangle\left\langle\beta_L'\right| = \sum_{\beta_L\beta_L'}(\psi\psi^*)_{\beta_L\beta_L'}\left|\beta_L\right\rangle\left\langle\beta_L'\right|$$

3, diagonalize ρ_L

$$\rho_L = \sum_{i=1}^{md}\omega_i\left|\omega_i\right\rangle\left\langle\omega_i\right|$$

and order the eigenvalues ω_i in descending order. If we assume the system to be left-right symmetric, we also have that $\rho_L = \rho_R$. Define the projector

$$\widehat{P} = \sum_{i=1}^{m}\left|\omega_i\right\rangle\left\langle\omega_i\right|$$

composed by only the first m eigenvalues of ρ_L

4. the projector P defines a truncation of the Hilbert space from md to states that can be used to compute the effective Hamiltonian of the system and all necessary operators in the reduced space, $\widetilde{\widehat{H}}_{L+1} = \widehat{P}^\dagger \widehat{H}_{L+1}\widehat{P}$ and given that $\widetilde{\widehat{H}}_{\text{int}} = \sum_k c_k A_L^k \otimes B_R^k$ the interaction term in the projected space can be computed applying the projector on the left and right separately. We thus obtain an effective matrix describing the Hamiltonian for the system of N sites of dimension m instead of md.

5. the algorithm is iterated starting again from step 1 provided that the Hamiltonian of the left and right block are replaced with the effective Hamiltonians computed in the previous step. The net effect is that one can describe a system of N+2 sites with a Hamiltonian of size (md)^2. At every step, the size of the described system is incremented by two sites while keeping the computational resources constant.

Part 2: Diagrammatic representation of step 1

And here we illustrate how step 1 works for XXZ model with S=1/2:

Example:

DMRG treatment of 1D XXZ model with S=1/2

$$H=\sum_i J(S_i^x S_{i+1}^x+S_i^y S_{i+1}^y)+J_z S_i^z S_{i+1}^z$$

Introduce ladder operator

$$S^+=S^x+iS^y$$

$$S^-=S^x-iS^y$$

$$S_i^+S_{i+1}^-+S_i^-S_{i+1}^+=2(S_i^x S_{i+1}^x+S_i^y S_{i+1}^y)$$

$$H=\frac{J}{2}\sum_i(S_i^+S_{i+1}^-+S_i^-S_{i+1}^+)+J_z\sum_i S_i^z S_{i+1}^z$$

Into matrix form $|\uparrow\rangle=\begin{pmatrix}1\\0\end{pmatrix},|\downarrow\rangle=\begin{pmatrix}0\\1\end{pmatrix}$

$$S^z=\frac{1}{2}\sigma^z=\begin{pmatrix}1/2 & \\ & -1/2\end{pmatrix},S^+=\frac{1}{2}\sigma^+=\begin{pmatrix} & 1\\ & \end{pmatrix},S^-=\frac{1}{2}\sigma^-=\begin{pmatrix} & \\ 1 & \end{pmatrix}$$

And the first step goes like:

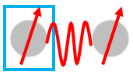
First consider the left block

block

$$\widehat{H}_L=\begin{pmatrix}0&0\\0&0\end{pmatrix}$$

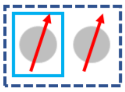
single site

$$\widehat{H}_1=\begin{pmatrix}0&0\\0&0\end{pmatrix}$$



interaction between
left block and site

$$\widehat{H}_{\text{L-int-1}}=\frac{J}{2}(S_L^+\otimes S_1^-+S_L^-\otimes S_1^+)+J_z S_L^z\otimes S_1^z=\begin{pmatrix}\frac{J_z}{4}&0&0&0\\0&-\frac{J_z}{4}&\frac{J}{2}&0\\0&\frac{J}{2}&-\frac{J_z}{4}&0\\0&0&0&\frac{J_z}{4}\end{pmatrix}$$



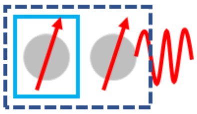
enlarged left block

$$\widehat{H}_{L+1} = \widehat{H}_{L} \otimes 1^{1} + \widehat{H}_{\mathrm{L-int-1}} + 1^{L} \otimes \widehat{H}_{1}$$

$$= \begin{pmatrix} 0 & \\ & 0 \end{pmatrix} \otimes 1 + \widehat{H}_{\mathrm{L-int-1}} + 1 \otimes \begin{pmatrix} 0 & \\ & 0 \end{pmatrix}$$

$$= \begin{pmatrix} 0 & & & \\ & 0 & & \\ & & 0 & \\ & & & 0 \end{pmatrix} + \widehat{H}_{\mathrm{L-int-1}} + \begin{pmatrix} 0 & & & \\ & 0 & & \\ & & 0 & \\ & & & 0 \end{pmatrix}$$

$$= \begin{pmatrix} \frac{J_z}{4} & 0 & 0 & 0 \\ 0 & -\frac{J_z}{4} & \frac{J}{2} & 0 \\ 0 & \frac{J}{2} & -\frac{J_z}{4} & 0 \\ 0 & 0 & 0 & \frac{J_z}{4} \end{pmatrix}$$



enlarged left block
interaction with others

$$S_{L+1}^{+} = 1_{L} \otimes S_{1}^{+} = \begin{pmatrix} & 1 & & \\ 0 & & & \\ & & & 1 \\ & & 0 & \end{pmatrix}$$

$$S_{L+1}^{-} = 1_{L} \otimes S_{1}^{-} = \begin{pmatrix} & 0 & & \\ 1 & & & \\ & & & 0 \\ & & 1 & \end{pmatrix}$$

$$S_{L+1}^{z} = 1_{L} \otimes S_{1}^{z} = \begin{pmatrix} \frac{1}{2} & & & \\ & -\frac{1}{2} & & \\ & & \frac{1}{2} & \\ & & & -\frac{1}{2} \end{pmatrix}$$

Do the same to the right block

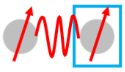
$$\widehat{H}_1 = \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix}$$

single site

$$\widehat{H}_R = \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix}$$

block

interaction between left block and site



$$\widehat{H}_{1\text{-int-R}} = \frac{J}{2}(S_1^+ \otimes S_R^- + S_1^- \otimes S_R^+) + J_z S_1^z \otimes S_R^z = \begin{pmatrix} \frac{J_z}{4} & 0 & 0 & 0 \\ 0 & -\frac{J_z}{4} & \frac{J}{2} & 0 \\ 0 & \frac{J}{2} & -\frac{J_z}{4} & 0 \\ 0 & 0 & 0 & \frac{J_z}{4} \end{pmatrix}$$

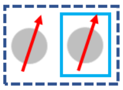
$$\widehat{H}_{1+R} = \widehat{H}_1 \otimes 1_R + \widehat{H}_{1\text{-int-R}} + 1_1 \otimes \widehat{H}_R$$

$$= 1 \otimes \begin{pmatrix} \quad & \quad \\ \quad & \quad \end{pmatrix} + \widehat{H}_{1\text{-int-R}} + \begin{pmatrix} \quad & \quad \\ \quad & \quad \end{pmatrix} \otimes 1$$

$$= \begin{pmatrix} \quad & \quad & \quad & \quad \\ \quad & \quad & \quad & \quad \\ \quad & \quad & \quad & \quad \\ \quad & \quad & \quad & \quad \end{pmatrix} + \widehat{H}_{1\text{-int-R}} + \begin{pmatrix} \quad & \quad & \quad & \quad \\ \quad & \quad & \quad & \quad \\ \quad & \quad & \quad & \quad \\ \quad & \quad & \quad & \quad \end{pmatrix}$$

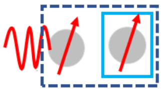
$$= \begin{pmatrix} \frac{J_z}{4} & 0 & 0 & 0 \\ 0 & -\frac{J_z}{4} & \frac{J}{2} & 0 \\ 0 & \frac{J}{2} & -\frac{J_z}{4} & 0 \\ 0 & 0 & 0 & \frac{J_z}{4} \end{pmatrix}$$

enlarged right block



$$S_{1+R}^+ = S_1^+ \otimes 1_R = \begin{pmatrix} & & 1 & \\ & & & 1 \\ 0 & & & \\ & 0 & & \end{pmatrix}$$

enlarged right block interaction with others

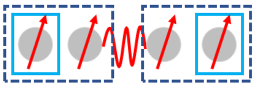


$$S_{1+R}^- = S_1^- \otimes 1_R = \begin{pmatrix} & & 0 & \\ & & & 0 \\ 1 & & & \\ & 1 & & \end{pmatrix}$$

$$S_{1+R}^z = S_1^z \otimes 1_R = \begin{pmatrix} \frac{1}{2} & & & \\ & \frac{1}{2} & & \\ & & -\frac{1}{2} & \\ & & & -\frac{1}{2} \end{pmatrix}$$

Now we need to treat the interaction between the left enlarged block and the right enlarged block:

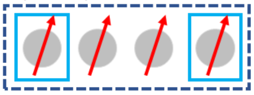
$$\hat{H}_{L+1-\mathrm{int}-1+R}=\frac{J}{2}(S^{+}_{L+1}\otimes S^{-}_{1+R}+S^{-}_{L+1}\otimes S^{+}_{1+R})+J_z S^{z}_{L+1}\otimes S^{z}_{1+R}=$$



interaction between
left enlarged block
and
right enlarged block

$$=\left(\begin{array}{cccccccccccccccc}
\frac{J_z}{4} & & & & 0 & 0 & 0 & 0 & & & & & & & & \\
 & \frac{J_z}{4} & & & 0 & 0 & 0 & 0 & & & & & & & & \\
 & & -\frac{J_z}{4} & & \frac{J}{2} & 0 & 0 & 0 & & & & & & & & \\
 & & & -\frac{J_z}{4} & 0 & \frac{J}{2} & 0 & 0 & & & & & & & & \\
0 & 0 & \frac{J}{2} & 0 & -\frac{J_z}{4} & & & & & & & & & & & \\
0 & 0 & 0 & \frac{J}{2} & & -\frac{J_z}{4} & & & & & & & & & & \\
0 & 0 & 0 & 0 & & & \frac{J_z}{4} & & & & & & & & & \\
0 & 0 & 0 & 0 & & & & \frac{J_z}{4} & & & & & & & & \\
 & & & & & & & & \frac{J_z}{4} & & & & 0 & 0 & 0 & 0 \\
 & & & & & & & & & \frac{J_z}{4} & & & 0 & 0 & 0 & 0 \\
 & & & & & & & & & & -\frac{J_z}{4} & & \frac{J}{2} & 0 & 0 & 0 \\
 & & & & & & & & & & & -\frac{J_z}{4} & 0 & \frac{J}{2} & 0 & 0 \\
 & & & & & & & & 0 & 0 & \frac{J}{2} & 0 & -\frac{J_z}{4} & & & \\
 & & & & & & & & 0 & 0 & 0 & \frac{J}{2} & & -\frac{J_z}{4} & & \\
 & & & & & & & & 0 & 0 & 0 & 0 & & & \frac{J_z}{4} & \\
 & & & & & & & & 0 & 0 & 0 & 0 & & & & \frac{J_z}{4}
\end{array}\right)$$

Finally we reach the last step in step 1:



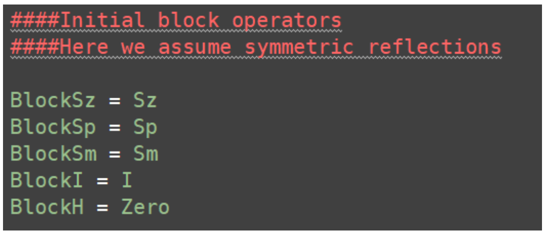
$$\hat{H}_4=\hat{H}_{L+1}\otimes 1_{1+R}+\hat{H}_{L+1-\mathrm{int}-1+R}+1_{L+1}\otimes\hat{H}_{R+1}=\begin{pmatrix}\frac{J_z}{4} & 0 & 0 & 0\\ 0 & -\frac{J_z}{4} & \frac{J}{2} & 0\\ 0 & \frac{J}{2} & -\frac{J_z}{4} & 0\\ 0 & 0 & 0 & \frac{J_z}{4}\end{pmatrix}\otimes 1_{1+R}+\hat{H}_{L+1-\mathrm{int}-1+R}+1_{L+1}\otimes\begin{pmatrix}\frac{J_z}{4} & 0 & 0 & 0\\ 0 & -\frac{J_z}{4} & \frac{J}{2} & 0\\ 0 & \frac{J}{2} & -\frac{J_z}{4} & 0\\ 0 & 0 & 0 & \frac{J_z}{4}\end{pmatrix}$$

$$=\left(\begin{array}{cccccccccccccccc}
\frac{3J_z}{4} & & & & 0 & 0 & 0 & 0 & & & & & & & & \\
 & \frac{J_z}{4} & \frac{J}{2} & & 0 & 0 & 0 & 0 & & & & & & & & \\
 & \frac{J}{2} & -\frac{J_z}{4} & & \frac{J}{2} & 0 & 0 & 0 & & & & & & & & \\
 & & & \frac{J_z}{4} & 0 & \frac{J}{2} & 0 & 0 & & & & & & & & \\
0 & 0 & \frac{J}{2} & 0 & -\frac{J_z}{4} & & & & \frac{J}{2} & & & & & & & \\
0 & 0 & 0 & \frac{J}{2} & & -\frac{3J_z}{4} & \frac{J}{2} & & & \frac{J}{2} & & & & & & \\
0 & 0 & 0 & 0 & & \frac{J}{2} & -\frac{J_z}{4} & & & & \frac{J}{2} & & & & & \\
0 & 0 & 0 & 0 & & & & \frac{J_z}{4} & & & & \frac{J}{2} & & & & \\
 & & & & \frac{J}{2} & & & & \frac{J_z}{4} & & & & 0 & 0 & 0 & 0 \\
 & & & & & \frac{J}{2} & & & & -\frac{J_z}{4} & \frac{J}{2} & & 0 & 0 & 0 & 0 \\
 & & & & & & \frac{J}{2} & & & \frac{J}{2} & -\frac{3J_z}{4} & & \frac{J}{2} & 0 & 0 & 0 \\
 & & & & & & & \frac{J}{2} & & & & -\frac{J_z}{4} & 0 & \frac{J}{2} & 0 & 0 \\
 & & & & & & & & 0 & 0 & \frac{J}{2} & 0 & \frac{J_z}{4} & & & \\
 & & & & & & & & 0 & 0 & 0 & \frac{J}{2} & & -\frac{J_z}{4} & \frac{J}{2} & \\
 & & & & & & & & 0 & 0 & 0 & 0 & & \frac{J}{2} & \frac{J_z}{4} & \\
 & & & & & & & & 0 & 0 & 0 & 0 & & & & \frac{3J_z}{4}
\end{array}\right)$$

Part 3: Plain code

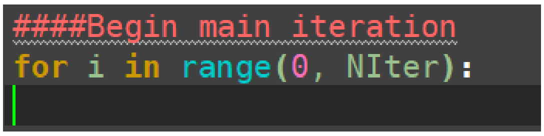
And we can write a plain code (code without any folding) with for-loop:

1, first step is to initial the single site Hamiltonian:



```
####Initial block operators
####Here we assume symmetric reflections

BlockSz = Sz
BlockSp = Sp
BlockSm = Sm
BlockI = I
BlockH = Zero
```

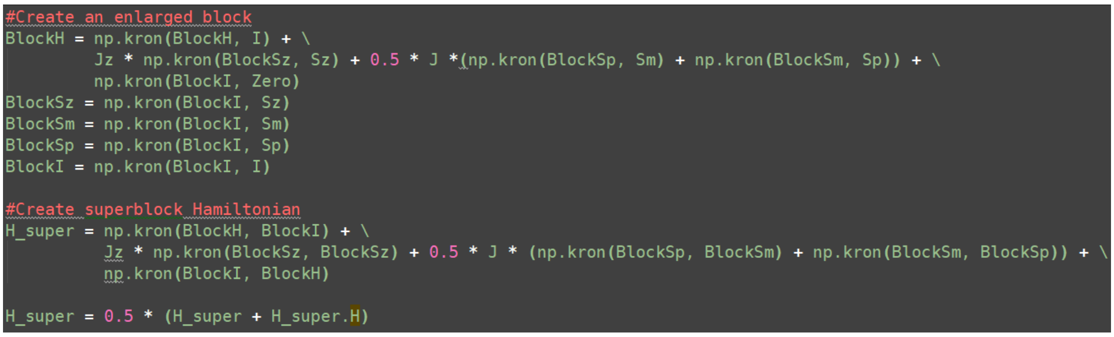
2, second step is to enter a for-loop:



```
####Begin main iteration
for i in range(0, NIter):
```

All the things related to the algorithm are done in this loop.

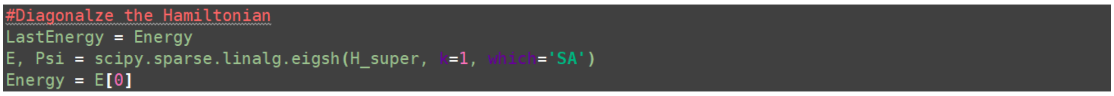
2.1, the first step has been shown in the above pictures, if translate into code, we have:



```
#Create an enlarged block
BlockH = np.kron(BlockH, I) + \
         Jz * np.kron(BlockSz, Sz) + 0.5 * J *(np.kron(BlockSp, Sm) + np.kron(BlockSm, Sp)) + \
         np.kron(BlockI, Zero)
BlockSz = np.kron(BlockI, Sz)
BlockSm = np.kron(BlockI, Sm)
BlockSp = np.kron(BlockI, Sp)
BlockI = np.kron(BlockI, I)

#Create superblock Hamiltonian
H_super = np.kron(BlockH, BlockI) + \
          Jz * np.kron(BlockSz, BlockSz) + 0.5 * J * (np.kron(BlockSp, BlockSm) + np.kron(BlockSm, BlockSp)) + \
          np.kron(BlockI, BlockH)

H_super = 0.5 * (H_super + H_super.H)
```

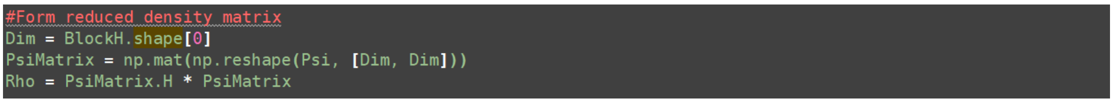
Here we have used the reflection symmetry in building the superblock Hamiltonian.

And solve the superblock Hamiltonian:



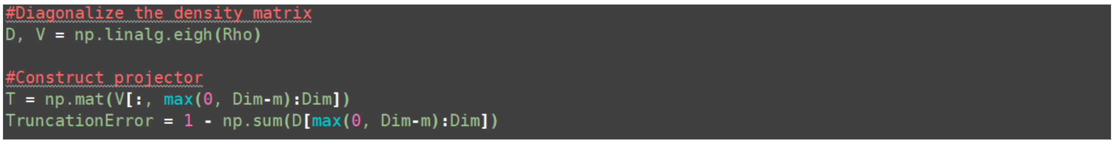
```
#Diagonalze the Hamiltonian
LastEnergy = Energy
E, Psi = scipy.sparse.linalg.eigsh(H_super, k=1, which='SA')
Energy = E[0]
```

2.2, now we can use the G.S. wavefunction to construct the density matrix of G.S.:



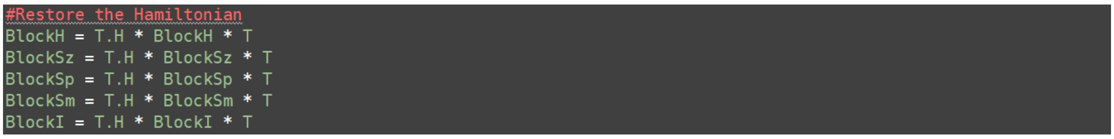
```
#Form reduced density matrix
Dim = BlockH.shape[0]
PsiMatrix = np.mat(np.reshape(Psi, [Dim, Dim]))
Rho = PsiMatrix.H * PsiMatrix
```

2.3, diagonalize reduced density matrix, make the truncation and build the projector:



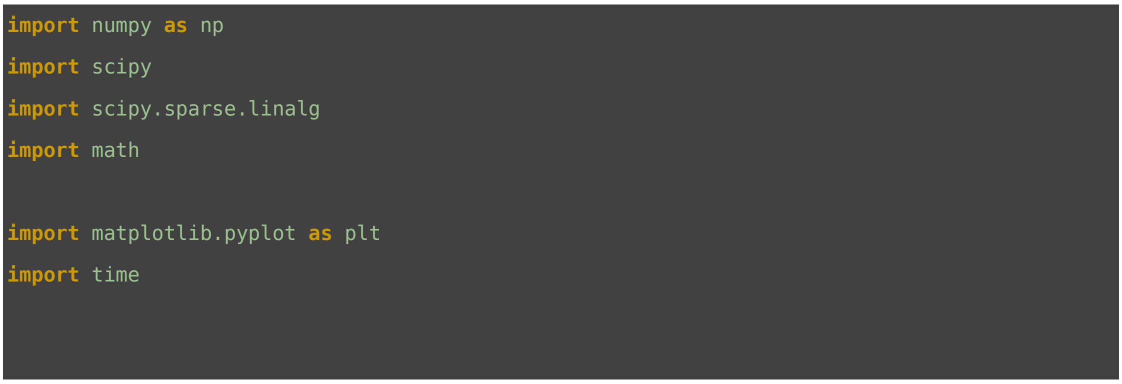
```
#Diagonalize the density matrix
D, V = np.linalg.eigh(Rho)

#Construct projector
T = np.mat(V[:, max(0, Dim-m):Dim])
TruncationError = 1 - np.sum(D[max(0, Dim-m):Dim])
```

2.4, restore the Hamiltonian and operators:



```
#Restore the Hamiltonian
BlockH = T.H * BlockH * T
BlockSz = T.H * BlockSz * T
BlockSp = T.H * BlockSp * T
BlockSm = T.H * BlockSm * T
BlockI = T.H * BlockI * T
```

So our code becomes:

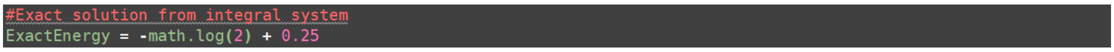


```
import numpy as np
import scipy
import scipy.sparse.linalg
import math


import matplotlib.pyplot as plt
import time
```

```
####Initial parameter
#physical parameter
J = 1
Jz = 1
#number of states kept
m = 10
#number of iterations
NIter = 100


#Open interactive mode
plt.ion()
plt.figure(1)

#Iinitalzie time axis
#t_now = 0

#Exact solution from integral system
ExactEnergy = -math.log(2) + 0.25
print("   Iter    Size    Energy     BondEnergy    EnergyError   Truncation")

####Local operator
I = np.mat(np.identity(2))
Sz = np.mat([[0.5, 0],
          [0, -0.5]])
Sp = np.mat([[0, 1],
           [0, 0]])
Sm = np.mat([[0, 0],
           [1, 0]])
Zero = np.mat(np.zeros((2, 2)))

####Initial block operators
####Here we assume symmetric reflections

BlockSz = Sz
BlockSp = Sp
BlockSm = Sm
BlockI = I
BlockH = Zero

Energy = -0.75

####Begin main iteration
for i in range(0, NIter):
    #Create an enlarged block
    BlockH = np.kron(BlockH, I) + \
            Jz * np.kron(BlockSz, Sz) + 0.5 * J *(np.kron(BlockSp, Sm) + np.kron(BlockSm, Sp))
+ np.kron(BlockI, Zero)
    BlockSz = np.kron(BlockI, Sz)
    BlockSm = np.kron(BlockI, Sm)
    BlockSp = np.kron(BlockI, Sp)
    BlockI = np.kron(BlockI, I)

    #Create superblock Hamiltonian
    H_super = np.kron(BlockH, BlockI) + \
             Jz * np.kron(BlockSz, BlockSz) + 0.5 * J * (np.kron(BlockSp, BlockSm) + np.kron(BlockSm,
BlockSp)) + np.kron(BlockI, BlockH)

    H_super = 0.5 * (H_super + H_super.H)

    #Diagonalze the Hamiltonian
    LastEnergy = Energy
    E, Psi = scipy.sparse.linalg.eigsh(H_super, k=1, which='SA')
    Energy = E[0]
    EnergyPerBond = (Energy - LastEnergy)/2

    #Form reduced density matrix
    Dim = BlockH.shape[0]
    PsiMatrix = np.mat(np.reshape(Psi, [Dim, Dim]))
    Rho = PsiMatrix.H * PsiMatrix

    #Diagonalize the density matrix
    D, V = np.linalg.eigh(Rho)

    #Construct projector
    T = np.mat(V[:, max(0, Dim-m):Dim])
    TruncationError = 1 - np.sum(D[max(0, Dim-m):Dim])

    print("{:6} {:6} {:16.8f} {:12.8f} {:12.8f} {:12.8f}".format(i, 4 + i * 2, Energy,
EnergyPerBond, ExactEnergy - EnergyPerBond, TruncationError))

    #Restore the Hamiltonian
    BlockH = T.H * BlockH * T
    BlockSz = T.H * BlockSz * T
    BlockSp = T.H * BlockSp * T
    BlockSm = T.H * BlockSm * T
    BlockI = T.H * BlockI * T
    t_now = i * 0.1
    plt.scatter(t_now, Energy)
    plt.pause(0.01)

print("Finished")
```

Several notes about the above code:

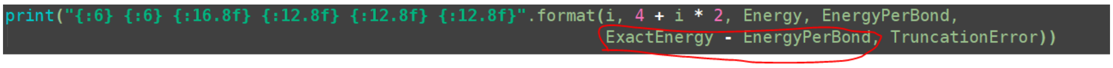
1, a benchmark result from integral system is given:



```
#Exact solution from integral system
ExactEnergy = -math.log(2) + 0.25
```

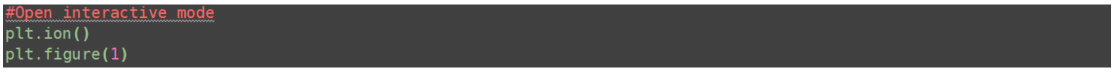
which gives the exact G.S. energy of 1D XXZ model with S=1/2.

And we have used the difference between exact G.S. energy and the calculated energy as a fingerprint:

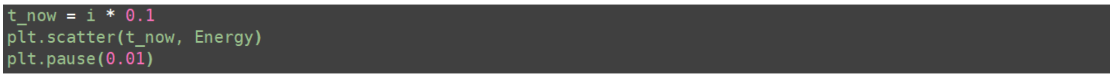


```
print("{:6} {:6} {:16.8f} {:12.8f} {:12.8f} {:12.8f}".format(i, 4 + i * 2, Energy, EnergyPerBond,
                                                              ExactEnergy - EnergyPerBond, TruncationError))
```

2, dynamical plot is used to see how energy evolves with respect to step:



```
#Open interactive mode
plt.ion()
plt.figure(1)
```

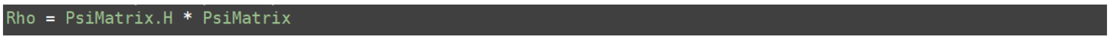


```
t_now = i * 0.1
plt.scatter(t_now, Energy)
plt.pause(0.01)
```

3, the main code is all about manipulation of matrix and tensor, see the related manual first for a better understanding, for example, the fusion and split of matrix is realized by:

np.mat(np.reshape(A,[n, m]))

The filling rule in np.reshape is row after row, therefore, the row d.o.f. of PsiMatrix comes from the left block and the column d.o.f. of PsiMatrix comes from the right block. Therefore, the following sum



```
Rho = PsiMatrix.H * PsiMatrix
```

will eliminate the d.o.f. of right block, resulting in the reduced density matrix for left block.

4, the above code is adapted from the following code written by Ian MaCulloch:

https://people.smp.uq.edu.au/IanMcCulloch/mptoolkit/index.php?n=Tutorials.SimpleDMRG?action=sourceblock&num=1

Part 4: Object orientated code

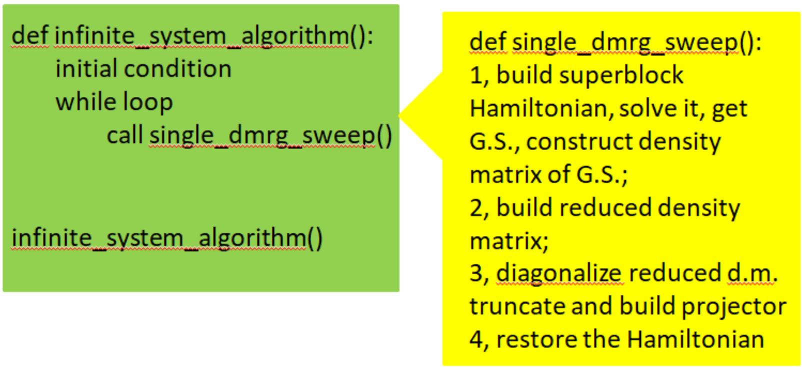
Plain code is good, but we want to write a more complex code by introduction of functions and optimize the code.

Basic thinking:

We can define a function to realize the function of for-loop in plain code, what we need is just calling that function:

```
def infinite_system_algorithm():
    initial condition
    while loop
        call single_dmrg_sweep()


infinite_system_algorithm()
```

This doesn’t solve the problem, but just fold the problem. We need to specify the steps in single_dmrg_sweep:

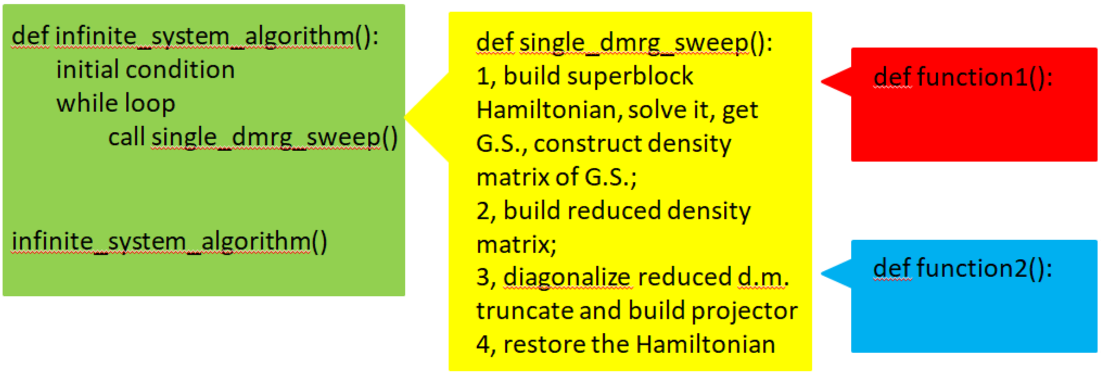


```
def infinite_system_algorithm():
    initial condition
    while loop
        call single_dmrg_sweep()


infinite_system_algorithm()
```

```
def single_dmrg_sweep():
1, build superblock Hamiltonian, solve it, get G.S., construct density matrix of G.S.;
2, build reduced density matrix;
3, diagonalize reduced d.m. truncate and build projector
4, restore the Hamiltonian
```

Note: this is iteration

What’s more, the basic step in single_dmrg_sweep can be divided into smaller pieces:



```
def infinite_system_algorithm():
    initial condition
    while loop
        call single_dmrg_sweep()


infinite_system_algorithm()
```

```
def single_dmrg_sweep():
1, build superblock Hamiltonian, solve it, get G.S., construct density matrix of G.S.;
2, build reduced density matrix;
3, diagonalize reduced d.m. truncate and build projector
4, restore the Hamiltonian
```

```
def function1():
```

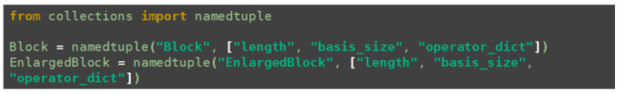
```
def function2():
```

First we define a special type to represent a block:

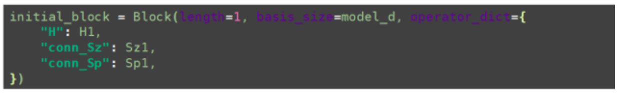
Size=1

Dimension=2

Operation: $\widehat{H}_L = \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix}$ $S^z = \begin{pmatrix} 1/2 & \\ & -1/2 \end{pmatrix}, S^+ = \begin{pmatrix} & 1 \\ & \end{pmatrix}$



```python
from collections import namedtuple

Block = namedtuple("Block", ["length", "basis_size", "operator_dict"])
EnlargedBlock = namedtuple("EnlargedBlock", ["length", "basis_size",
"operator_dict"])
```

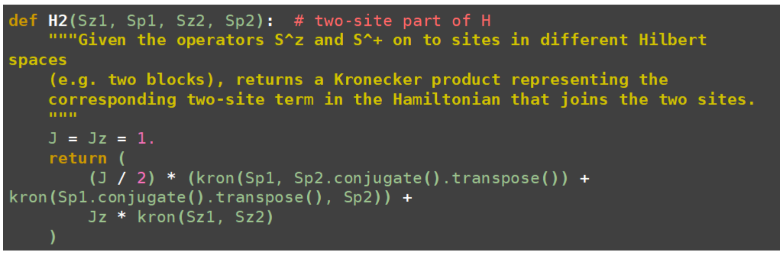


```python
initial_block = Block(length=1, basis_size=model_d, operator_dict={
    "H": H1,
    "conn_Sz": Sz1,
    "conn_Sp": Sp1,
})
```

namedtuple is a subclass of tuple. Here we use namedtuple to characterize the block, where the three attributes: length, basis_size and operator_dict directly correspond to size, dimension and operations of the block.

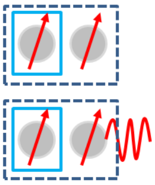
Second we define several functions to perform the basic steps:
1, define a function to construct interaction between two operators:

$$H=\frac{J}{2}(S_i^+S_{i+1}^-+S_i^-S_{i+1}^+)+J_zS_i^zS_{i+1}^z$$



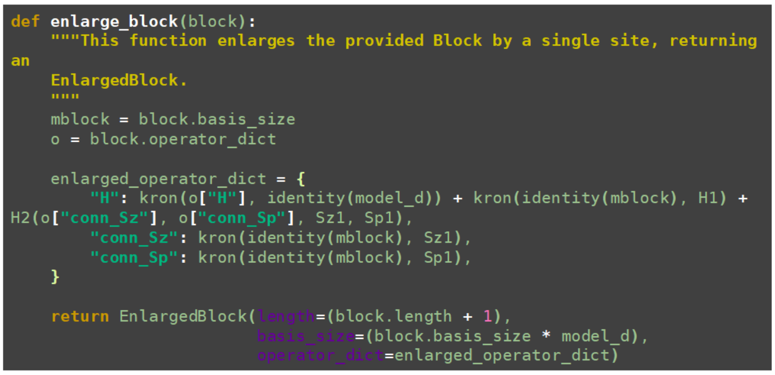
```
def H2(Sz1, Sp1, Sz2, Sp2):  # two-site part of H
    """Given the operators S^z and S^+ on to sites in different Hilbert
spaces
    (e.g. two blocks), returns a Kronecker product representing the
    corresponding two-site term in the Hamiltonian that joins the two sites.
    """
    J = Jz = 1.
    return (
        (J / 2) * (kron(Sp1, Sp2.conjugate().transpose()) +
kron(Sp1.conjugate().transpose(), Sp2)) +
        Jz * kron(Sz1, Sz2)
    )
```

2, define a function to turn a block into an enlarged block:



$$\widehat{H}_{L+1}=\widehat{H}_L\otimes 1^1+\widehat{H}_{L-\mathrm{int}-1}+1^L\otimes\widehat{H}_1$$

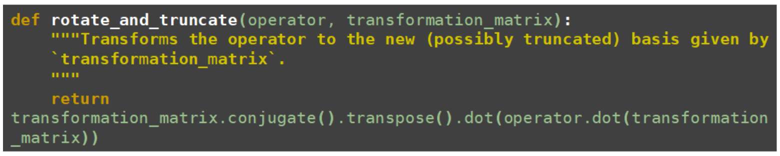
$$S_{L+1}^z=1_L\otimes S_1^z \qquad S_{L+1}^+=1_L\otimes S_1^+$$



```
def enlarge_block(block):
    """This function enlarges the provided Block by a single site, returning
an
    EnlargedBlock.
    """
    mblock = block.basis_size
    o = block.operator_dict

    enlarged_operator_dict = {
        "H": kron(o["H"], identity(model_d)) + kron(identity(mblock), H1) +
H2(o["conn_Sz"], o["conn_Sp"], Sz1, Sp1),
        "conn_Sz": kron(identity(mblock), Sz1),
        "conn_Sp": kron(identity(mblock), Sp1),
    }

    return EnlargedBlock(length=(block.length + 1),
                         basis_size=(block.basis_size * model_d),
                         operator_dict=enlarged_operator_dict)
```

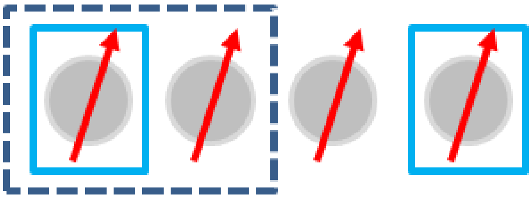
3, define a function to truncate

$$\hat{\hat{H}}_{L+1}=\hat{P}^{\dagger}\hat{H}_{L+1}\hat{P}$$

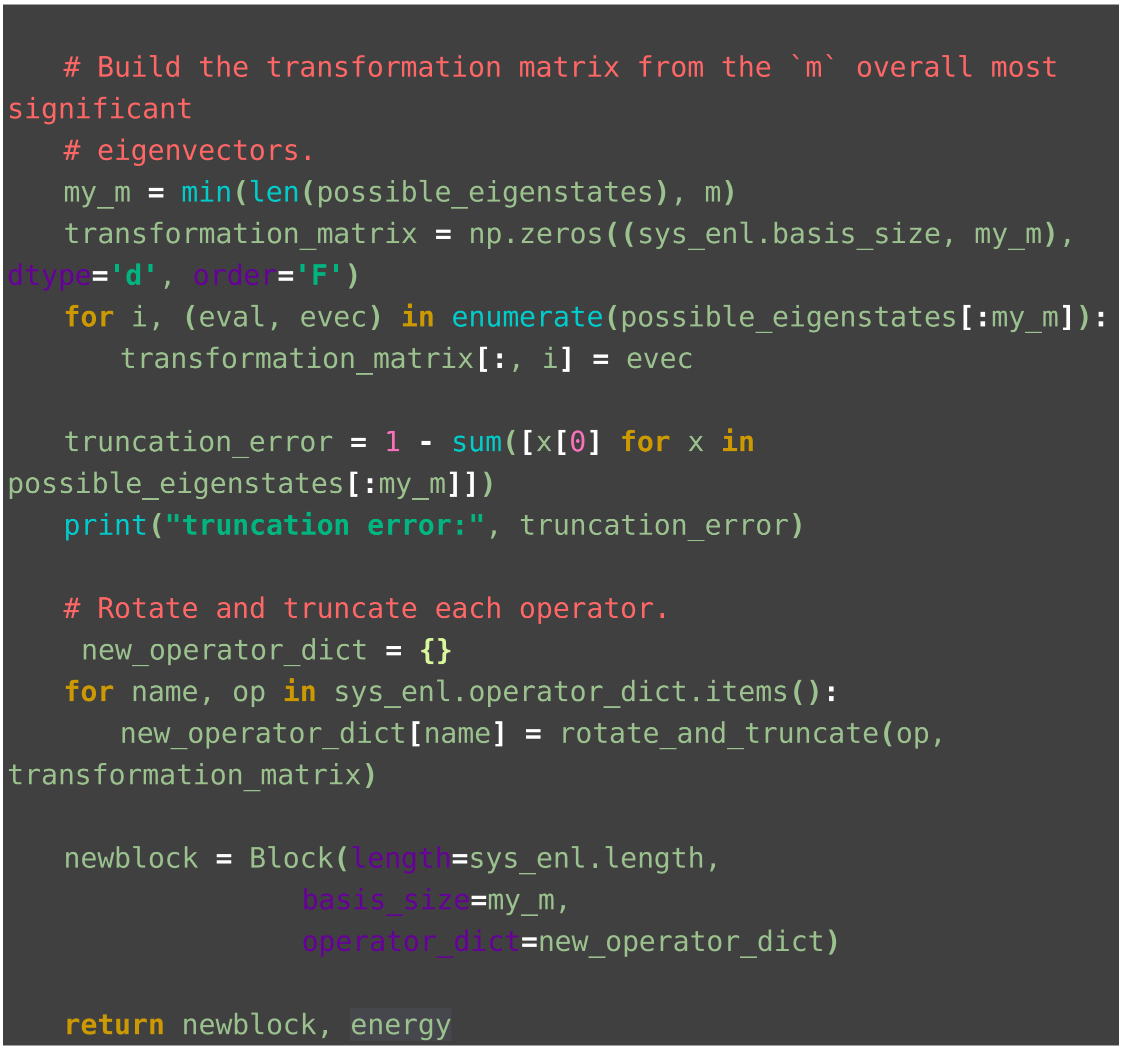


```
def rotate_and_truncate(operator, transformation_matrix):
    """Transforms the operator to the new (possibly truncated) basis given by
    `transformation_matrix`.
    """
    return
transformation_matrix.conjugate().transpose().dot(operator.dot(transformation
_matrix))
```

Now we can define a single_dmrg_step. Here we input sys, env and the state kept, since the single_dmrg_step is iteration, output should a newblock and energy:

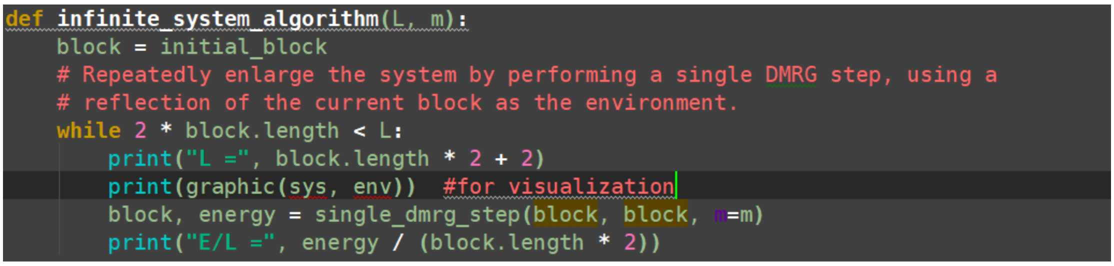


```
def single_dmrg_step(sys, env, m):
    sys_enl = enlarge_block(sys)
    if sys is env:  # no need to recalculate a second time
        env_enl = sys_enl
    else:
        env_enl = enlarge_block(env)

    # Construct the full superblock Hamiltonian.
    m_sys_enl = sys_enl.basis_size
    m_env_enl = env_enl.basis_size
    sys_enl_op = sys_enl.operator_dict
    env_enl_op = env_enl.operator_dict
    superblock_hamiltonian = kron(sys_enl_op["H"],
identity(m_env_enl)) + kron(identity(m_sys_enl), env_enl_op["H"]) +
                             H2(sys_enl_op["conn_Sz"],
sys_enl_op["conn_Sp"], env_enl_op["conn_Sz"],
env_enl_op["conn_Sp"])

    # Call ARPACK to find the superblock ground state.  ("SA" means
find the
    # "smallest in amplitude" eigenvalue.)
    energy, psi0 = eigsh(superblock_hamiltonian, k=1, which="SA")

    # Construct the reduced density matrix of the system by tracing
out the
    # environment
    psi0 = psi0.reshape([sys_enl.basis_size, -1], order="C")
    rho = np.dot(psi0, psi0.conjugate().transpose())

    # Diagonalize the reduced density matrix and sort the eigenvectors
by
    # eigenvalue.
    evals, evecs = np.linalg.eigh(rho)
    possible_eigenstates = []
    for eval, evec in zip(evals, evecs.transpose()):
        possible_eigenstates.append((eval, evec))
    possible_eigenstates.sort(reverse=True, key=lambda x: x[0])  #
largest eigenvalue first
```



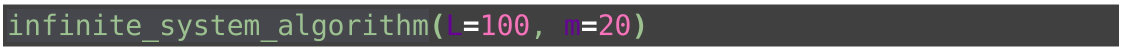
```
    # Build the transformation matrix from the `m` overall most
significant
    # eigenvectors.
    my_m = min(len(possible_eigenstates), m)
    transformation_matrix = np.zeros((sys_enl.basis_size, my_m),
dtype='d', order='F')
    for i, (eval, evec) in enumerate(possible_eigenstates[:my_m]):
        transformation_matrix[:, i] = evec

    truncation_error = 1 - sum([x[0] for x in
possible_eigenstates[:my_m]])
    print("truncation error:", truncation_error)

    # Rotate and truncate each operator.
     new_operator_dict = {}
    for name, op in sys_enl.operator_dict.items():
        new_operator_dict[name] = rotate_and_truncate(op,
transformation_matrix)

    newblock = Block(length=sys_enl.length,
                     basis_size=my_m,
                     operator_dict=new_operator_dict)

    return newblock, energy
```

Finally we can define infinite_system_algorithm



```
def infinite_system_algorithm(L, m):
    block = initial_block
    # Repeatedly enlarge the system by performing a single DMRG step, using a
    # reflection of the current block as the environment.
    while 2 * block.length < L:
        print("L =", block.length * 2 + 2)
        print(graphic(sys, env))  #for visualization
        block, energy = single_dmrg_step(block, block, m=m)
        print("E/L =", energy / (block.length * 2))
```

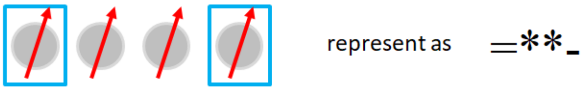
And what we need to do is just call the algorithm:

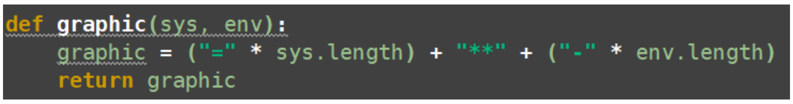


```
infinite_system_algorithm(L=100, m=20)
```

Visualization:

To see how the system growth, we use the following correspondence:





```
def graphic(sys, env):
    graphic = ("=" * sys.length) + "**" + ("-" * env.length)
    return graphic
```

Notes:

1, In part 3, we use the package math and transform the operator into matrix;

In part 4, each operator is just array. So the operator manipulations are different.